

Corrigio
Ezequiel

Organización del Computador 2

Segundo parcial – 15/11/18

Normas generales

1 (33)	2 (40)	3 (33)	
12,5	35	30,5	= 8 (A)

- Numere las hojas entregadas. Complete en la primera hoja la cantidad total de hojas entregadas.
- Entregue esta hoja junto al examen, la misma **no** se incluye en la cantidad total de hojas entregadas.
- Esta permitido tener los manuales y los apuntes con las listas de instrucciones en el examen. Está prohibido compartir manuales o apuntes entre alumnos durante el examen.
- Cada ejercicio debe realizarse en hojas separadas y numeradas. Debe identificarse cada hoja con nombre, apellido y L.U.
- La devolución de los exámenes corregidos es personal. Los pedidos de revisión se realizarán por escrito, antes de retirar el examen corregido del aula.
- Los parciales tienen tres notas. I (Insuficiente): 0 a 59 pts, A- (Aprobado condicional) 60 a 64 pts y A (Aprobado) 65 a 100 pts. No se puede aprobar con A- ambos parciales. Los recuperatorios tienen dos notas I 0 a 64 pts y A 65 a 100 pts.

Ej. 1. (25 puntos)

- (25p) a. Considerando la siguiente tabla de traducciones de direcciones por segmentación y paginación. De ser posible, dar un conjunto de descriptores de segmento, directorio de paginas y tablas de paginas que cumplan con todas las traducciones **simultáneamente**. Detallar los campos de todas las estructuras involucradas. Además indicar desde que segmento de código se esta ejecutando cada acceso, si la traducción es *identity mapping* y en el caso que alguna traducción no sea posible indicar, ¿por qué?

Lógica	Lineal	Física	Características
0x0193:0x836401A7	0x960C31A7	0x332A21A7	Lectura de 4 bytes como nivel 0 a nivel 3
0x01A0:0x0392412A	0x960C312A	0x332A212A	Lectura de 2 bytes, como nivel 0 a nivel 0, solo lectura.
0x02A8:0x834AAFFF	0x8397AFFF	0x8A9FEFFF	Escritura de 4 bytes, como nivel 0 a nivel 0
0x02E2:0x01B10FFF	0x01B10FFF	0x0010FFFF	Escritura de 1 byte, como nivel 2 a nivel 2

Ej. 2. (40 puntos)

Se crea un sistema con segmentación flat y paginación activa, en dos niveles de protección, que ejecuta concurrentemente un conjunto de hasta 1024 tareas independientes. Estas tienen acceso al servicio **New-Task**, que crea una instancia de tarea de entre tres posibles códigos almacenados en el sistema. El servicio toma como parámetro en el registro **eax** un número del 1 al 3 que indica cuál es el código a cargar en la nueva tarea. Además, en este mismo registro se debe retornar uno de los siguientes valores:

- 1 a 3: La nueva instancia de tarea fue creada, indicando de qué tipo fue.
- -1: Ninguna tarea nueva fue creada, indicando que un error.

Considerar que los registros restantes no deberán ser alterados por la ejecución del servicio. En el caso de la nueva tarea, esta deberá almacenar en **edx** el id de la tarea que la creo

- (10p) a. Definir un posible mapa de memoria. Indicar el rango de direcciones de:

- paginas de kernel
- paginas de código y datos de las tareas. Indicar su tamaño.
- paginas donde se mapean las tareas creadas.

Además, indicar que información específica es almacenada en cada rango designado. Por ejemplo: *Page Directory*. Escribir el esquema de paginación utilizado detallando el mapeo de cada área de memoria junto con sus atributos. Preferentemente realizar un dibujo del mismo.

- (7p) b. Definir las entradas en la IDT para el servicio `NewTask`, para las rutinas de excepciones y para la interrupción de reloj. Complete todos los campos necesarios.
- (24p) c. Implementar en ASM y/o C la rutina del servicio `NewTask`.

Se cuenta con las siguientes funciones:

- `tss* tss_getFree()`: Retorna una `tss` libre para una nueva tarea. Si no existe `tss` libre retornará `null`.
- `uint32_t sched_getId()`: Retorna el id de la tarea actual.
- `void sched_add(tss*)`: Agregar un puntero a una `tss` dentro del scheduler para ser ejecutada.
- `pde* mmu_newPD()`: Retorna un directorio de paginas donde todas sus entradas son no presente.
- `void mmu_mapPage(pde* cr3, void* virtual, void* fisica, uint8_t us, uint8_t rw)`: Mapea la pagina virtual al marco de pagina fisica en el mapa de memoria dado por `cr3` con los atributos `us` y `rw`.

Ej. 3. (35 puntos)

Se desea implementar una funcionalidad de kernel que cada vez que se desaloje una tarea dentro de la rutina de atención de interrupciones del reloj, se almacene en que función del código de la tarea se produjo la interrupción.

- (10p) a. Suponiendo que se cuenta con una función que indica la próxima tarea a ejecutar. Construir una posible rutina de atención de interrupciones de reloj que utilice dicha función para intercambiar tareas. Explicar el funcionamiento de la rutina y de la función que indica la próxima tarea ¿Qué pasa en el caso que el intercambio de tareas sea por la misma tarea?
- (25p) b. Modificar la rutina anterior para agregar la funcionalidad pedida. Considerar que se cuenta con la función `void logEIP(uint32_t gdtIndex, void* func)`, que toma el índice en la GDT de la tarea que se encontraba ejecutando y el puntero a la función que se estaba ejecutando.

Nota: Asumir que todas las subrutinas dentro del código fueron llamadas mediante la instrucción `call`. Esta ocupa exactamente 6 bytes, siendo los últimos 4 bytes la dirección de la función a llamar.

1) Primeros campos que índices de la GDT deben estar presentes. ~~Los~~ Veamos los 4

$0x0173 = 0001\ 0001\ 1001\ 0011 \rightarrow RPL = 11$ Índice de selector = $0x32$
 $0x01A0 = \dots \dots \dots \dots \rightarrow RPL = 00$ " " " " = $0x34$
 $0x02A8 = \dots \dots \dots \dots \rightarrow RPL = 00$ " " " " = $0x55$
 $0x02B2 = \dots \dots \dots \dots \rightarrow RPL = 10$ " " " " = $0x56$
 (también están en la GDT)

Veamos qué pasa con las direcciones lineales:

$0x960C31A7 \rightarrow Dir = 10\ 0101\ 1000 = 0x258$
 Table = $0x0C3$
 Offset = $0x1A7$

$0x0392412A \rightarrow Dir = 10\ 0000\ 1101 = 0x00E$
 Table = $0x000$ Offset = $0x12A$
 $0x834AAFF7 \rightarrow Dir = 10\ 0000\ 1101 = 0x20D$
 Table = $0x0AA$ Offset = $0xFF7$

El 2º tiene la misma Dir y Table. Solo cambia el offset

$0x8317AFF7 \rightarrow Dir = 10\ 0000\ 1101 = 0x20E$
 T = $01\ 0111\ 1010 = 0x17A$
 Off = $0xFF7$

$0x01B10FFF \rightarrow Dir = 10\ 0000\ 0110 = 0x006$
 Table = $11\ 0001\ 0000 = 0x310$
 Offset = $0xFFF$

Esto significa que la tabla de directorios tiene que tener presentes los entornos $0x258$ (que direcciona a una tabla con el índice $0x0C3$ presente), $0x20E$ (con índice de PT $0x17A$ presente), $0x20D$ (con índice de PT $0x17A$ presente) y $0x006$ (con índice de PT $0x310$ presente).

A su vez, las 2 primeras direcciones se mapean a la misma página cuya base es $0x332A2$. Las otras 2 mapean las bases $0x8A7FE$ y $0x0010E$ respectivamente.

Además, falta la base y el límite de los segmentos.

queremos que $Linear = base + lógica$ y que $lógica < límite$

$\Rightarrow base = linear - lógica$
 $0x960C31A7 - 0x12A83000 = 0x834AAFF7$
 $0x834AAFF7 - 0x004D0000 = 0x8317AFF7$

$0x960C31A7 - 0x12A83000 = 0x834AAFF7$
 $0x834AAFF7 - 0x004D0000 = 0x8317AFF7$

No se cumple $lógica < límite!!$

Ahora podemos armar GDT y esquema de paginación:

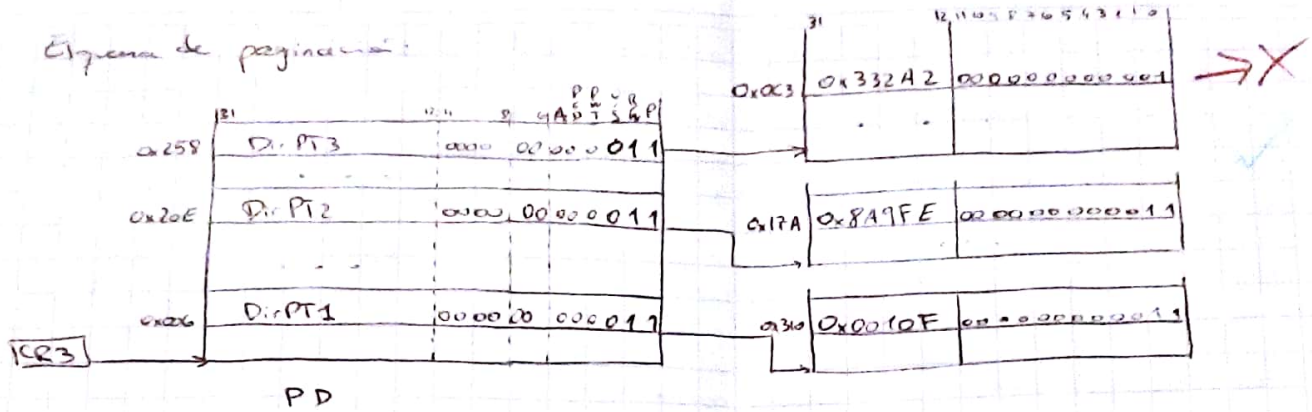
	Indr	Base	Límite	G	D	A	P	D	S	Tree
	0	descriptores nulos: todos los bits a 0								
	...									
	0x32	0x12A83000	0xFFFFF	0	1	0	0	1	0	0xE
	0x33	No importa. No								
	0x34	0x9277F000	0xFFFFF	0	1	0	0	1	0	0x0
	...									
	0x55	0x004D0000	0xFFFFF	0	1	0	0	1	0	0x2
	0x56	0x00000000	0xFFFFF	0	1	0	0	1	2	0x2

Gdtr = 0x00000000

containing 1 page
it's 0x30000000

Los ponemos algo lo suficientemente grande como para que entre por lo menos 4096 bytes

Esquema de paginación:



Dir PT1..3 son direcciones de memoria física que son la base de las PT. Pueden ser cambiadas, solo deben no pisarse a sí mismas ni a las 4 direcciones del cruceado (y estar dentro de la memoria, claro)

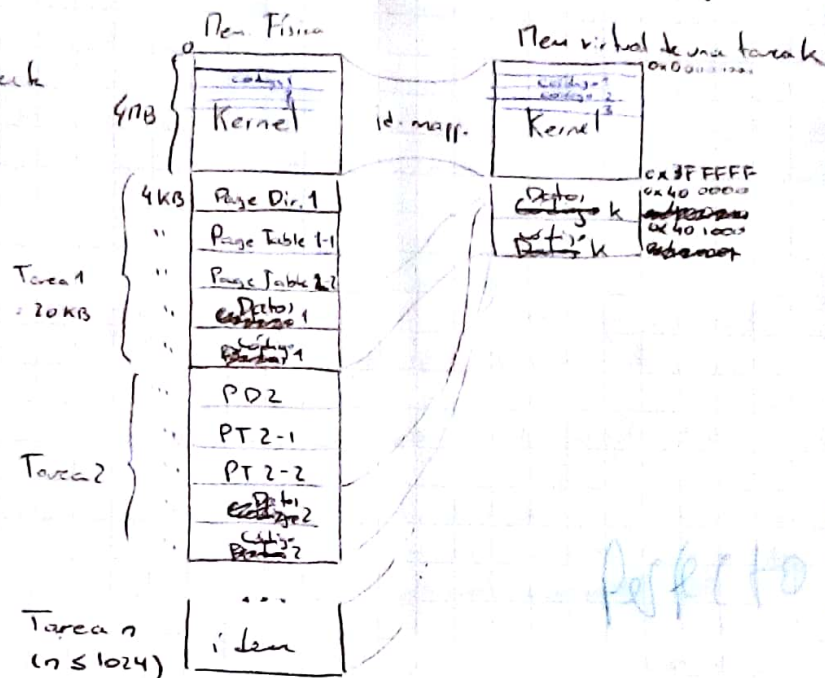
- Ejemplo de código nivel 0 pero con $RPL = 3$ (correcto, $CPL = 0$)
El EPL no importa porque es constante
- Ejemplo con $CPL = 0$ y $RPL = 0 \Rightarrow EPL = 0$
- " " $CPL = 0$ y $RPL = 0 \Rightarrow EPL = 0$
- " " $CPL = 2$ y $RPL = 2 \Rightarrow EPL = 2$

Ninguna traducción es identity mapping.

→ X Las entradas 1 y 2 no pueden estar definidas al mismo tiempo, y a que no pudes hacer que los permisos sean usuario y supervisor AL MISMO TIEMPO, se esperaba que identificaran este problema, y eligieran una de las entradas para no definir la

2) a. Para el kernel voy a usar ~~4MB~~ ^{4MB} los primeros 4MB de memoria donde, entre otras cosas, voy a tener guardada ~~3~~ ³ páginas para cada uno de los 3 códigos (1 para el kernel. Al kernel lo mapeo con identity mapping. Luego, para cada tarea voy a usar (además de una página para la PD) una página para el código, una para ~~datos~~ ^{datos} (donde también ponga a las pilas) y 2 para ~~las~~ ^{las} PTs (en una voy a estar las entradas asociadas al kernel y en la otra, las asociadas a otras áreas páginas).

¡Te queda lo siguiente!



Perfecto!

Las nuevas tareas creadas se van mapeando en la siguiente página libre que se tenga disponible. Las TSSs están dentro del kernel.

Atributos: Las páginas del kernel tendrán $U/S = 0$ (tanto a la entrada de la PD como en las 1024 de la PT asociada). $R/W = 0$ (no queremos escribir en el kernel, salvo llegar el caso, en la pila o lo que la situación) (así de código y datos), $U/S = 1$ y $R/W = 1$ para ~~datos y código~~ ^{datos y código} (que no se escriba si es código se encarga la unidad de segmentación).

En todos los casos, el bit de Presente está seteado. El resto de los campos, en 0, salvo la base del page frame que para los del kernel será igual al índice de su PTE y para las tareas, le corresponderá su número +3 y +4 para ~~código y datos~~ ^{código y datos} respectivamente. Los address de las PT siguen la misma lógica pero +1 y +2.

b. Las entradas de las excepciones serán Interrupt Gates (I.G.) ~~identificadas~~ ^{identificadas} ocupando los índices 0 a 31. La interrupción del reloj (también I.G.) ocupará la entrada 32 y la syscall la pondremos en 0x80 (en honor a Linux), también como I.G.

De este modo, las I.G. de las entradas 0 a 32 y la 0x80 tendrán $P=1$, $DPL=0$ (salvo int 0x80 con $DPL=3$ para poder ser llamado por las tareas), $D=1$ y el resto de los bits en 0 excepto:

- bits 9 y 10 (que identifican que sea I.G.)
- Offset: ~~guarda~~ ^{guarda} apuntará al lugar del código donde empieza el handler de la interrupción asociada.
- Selector de segmento: será el del segmento de código nivel 0.

(*) Además, me voy a guardar los 3 códigos posibles de las tareas en las páginas 1, 2 y 3 del kernel respectivamente. Así es más fácil encontrarlas para copiarlas luego.

(*) Esa será la vía PTE con $R/W=1$. De todas maneras, es seguro pues accederemos desde el kernel si llegamos a usarla.

C. Asumo que el scheduler maneja a las tareas con un vector `tareas[17]`.

ISR-80:

```

pushad
push eax
call NewTask_C ; en eax retorna 1, 3 o -1
sub esp, 4
cmp eax, -1
je error
popad ; eax tiene el mismo valor que retornó pues, si pedí copiar código 2, p.ej, NewTask.C
iret ; me voy a devolver en eax un 2 así que es lo mismo.

```

error: popad

mov eax, -1

iret

→ Pero de alguna forma te tenés que encargar del PUSHAD
es lo así como lo pusiste no anda

Lo que vamos a hacer en la función de C es:

- 1) ver si ~~podemos~~ hay menos de 1024 tareas. Si no se puede agregar, retornamos -1
- 2) Llamar a `tss_getFree()`. Si retorna NULL, retornamos -1
- 3) Creamos una nueva PD. ~~Representamos la 2p~~
- 4) ~~Representamos el kernel~~
- 5) ~~Representamos el código~~
- 6) ~~Representamos la nueva TSS~~
- 7) ~~Completamos la nueva~~ Copiamos el código
- 8) ~~Representamos la nueva TSS~~
- 9) Agregamos TSS al scheduler
- 10) Retornamos con el mismo parámetro de entrada

```

uint32_t NewTask (uint32_t numeroCodigo) {
    uint32_t id-actual = sched-getId();
    if (tareas.size() == 1024) return -1; // Para 1
    tss * nuevaTSS = tss-getFree();
    if (nuevaTSS == NULL) return -1; // Para 2
    pde * nuevaPD = mmu-newPDL(); // 3
    for (int i=0; i<1024; ++i) { // 4
        mmu-mapPage (nuevaPD, i*4096, i*4096, 0, 1);
    }
    mmu-mapPage (nuevaPD, 0x40000, 0x40000 + id-actual * 5*4096 + 3*4096, 1, 1); // 5
    mmu-mapPage (nuevaPD, 0x40000, 0x40000 + id-actual * 5*4096 + 4*4096, 1, 1); // 6
    copiar copiarCodigo (numeroCodigo, 0x40000 + id-actual * 5*4096 + 4*4096);
}

```

nuevaTSS → iompg = 0x FFFF

" → CS = ~~0x~~ SELECTOR-SEGMENTO-usuario-UV13;

" → DS = " " DATOS

" → FS = "

" → SS = "

" → ES = "

" → SS = "

// Dependen de cómo se arma la GDT

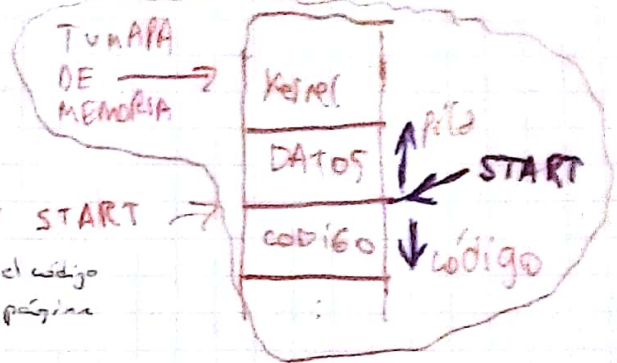
// pero como que se sabe que los datos van a ser

↓
No me pidieron hablar de segmentación, pero esto no lo hice, pero se usaba una GDT con un descriptor nulo, 2 de código, 2 de datos (uno nulo, 3 de usuario) y luego los descriptors de los TSS para ponerlos en los Casos de saber cuál es el selector

nuevaTSS → edi = id-actual; → esto lo en edx, pero se entiende la intención

```

    → ei = 0;
    → ebx = 0;
    → edx = 0;
    → ecx = 0;
    → eax = 0;
    → eflags = 0x202;
    → cr3 = nuevaPD;
    → eip = START START; // donde empieza el código
    → esp = 0xFFF; // al final de la página
    → ss0 = SEG_1EG-DATOS-LVLO;
    → esp0 = 0xFFF
  
```



Nota que para las pilas de nivel 1 y 2 no me importan sus usuarios. Solo 2 niveles de protección. Lo mismo con el selector de segmento de la LDT. Como no la usamos, no sirve.

`sched-add(nuevaTSS);`
`return numeroCodigo;`

`void copiarCodigo(uint32_t numero, uint32_t* dst)`

`uint32_t * src = 4096 * numero;` // asumo que guardé el código de 1..3 en las páginas
 // del kernel 1 a 3, así es más fácil encontrar donde

`for (int i=0; i<4096; ++i) {`

`dst[i] = src[i];` → **comparoo! no tenés mapeada esta página**

física, no podés copiar esto, deberías haberle mapeado la página previamente a tu propio directorio (o sea al CR3 actual) (como en el TP2).

3) a. Llamo a la función dada sched-proxima-tarea y me devuelve el selector de la misma.

tss-selector dw 0
tss-offset dd 0 } → esto definido al revés de como debería ir

ISR-32:

```
pushad
call ret fin-int-pte
call sched-proxima-tarea
str bx
cmp ax, bx
je .fin (*)
mov [tss-selector], ax
jmp far [tss-offset]

.fin:
popad
iret
```

En primer lugar, ~~tenemos~~ pushad todos los registros, pues llamo a 2 funciones y capaz que cambio de tarea, así que no quiero perder nada.

El ~~programa~~ le aviso al pic que atender la interrupción del reloj.

Llamo a la función dada que lo que hace es fijarse en el scheduler cuál es la próxima tarea y me devuelve su selector (normalmente construimos el scheduler con un vector de que contiene a las tareas activas).

Si cargamos en bx el selector de la tarea actual.

NO ES ESTE EL MOTIVO

Cargamos con la próxima tarea: si son la misma, no hace falta cambiar (pues comparamos en la misma, es redundante). En tal caso, ya podemos volver a la tarea habiendo recuperado previamente los registros de la pila.

Sino, movemos este nuevo selector al lugar de ~~memoria~~ donde está el selector para luego hacer un jmp far con el offset dado.

b. Antes de escribir el código, expliquemos lo que vamos a hacer: ~~se debe hacer~~

Al hacer un call común y corriente, se guarda en la pila el ~~valor~~ EIP de retorno. Si conseguimos esto, podemos ir a esa dirección, ~~mirar~~ mirar los 4 bytes anteriores y obtener, ¿quién es la función?

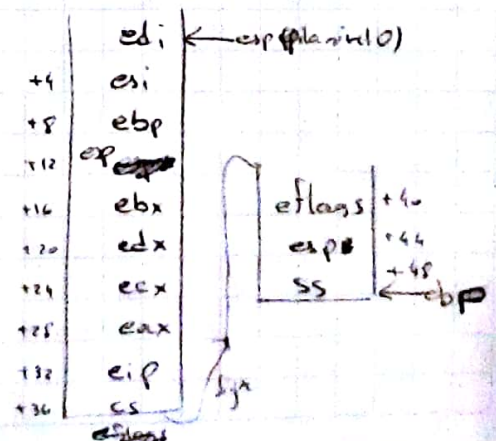
Pero, ¿dónde está este EIP? Asumiendo que detrás de esto hubo programadores decentes, se ha armado el stack frame al caer en la última función, por lo que el EBP apunta al EBP de la función llamadora que está inmediatamente por encima (por debajo pues es expand down) del EIP buscado.

Pero luego vino la interrupción del reloj y un cambio de ~~privilegio~~ ^{privilegio}. Esto significa que no podemos acceder tan fácil a esto. Tenemos que buscar el SS y EBP en la pila de nivel 0 para luego ir a la vieja pila de nivel 3 donde está lo que queremos.

Luego del pushad me queda la pila de la siguiente manera:

→ en [esp+44] tengo el esp original
→ en [esp+48] " " " original

→ en [esp+8] tengo el ebp que quería
y en [esp+48] el SS original



→ lo que tengo estado lo mismo hasta (*) y luego:

```

je fin (*)
mov ebx, [esp+8] ; ebx = ebp-ult
mov edx, [esp+48] ; eax = SS
mov eax, [eax:ebx] ; largo de esp que es (ebp-ultima)
mov ebx, [eax]

```

```

mov ecx, [edx:ebx+4]
mov edx, [esp+36] ; eax = selector de seg. del código original
mov ebx, [edx:eip-4] ; ebx = *func que quise

```

; Finalmente quiero el índice en la GDT, o sea, no quiero los 10-3 bits del selector:

```

shr edx, 3 ; ahora sí tengo el índice

```

```

push eax
push edx
push ebx

```

```

call logEIP

```

```

add esp, 8

```

```

pop eax

```

```

mov [tss-selector], eax

```

```

jmp far [tss-offset]

```

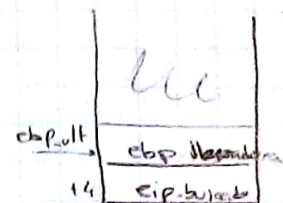
fin:

```

popad
iret

```

pila original:



Función llamada:

ep → cell última función por reloj

→ Querés el índice de la tarea, o sea de la TSS, NO del selector de código. La idea es que uses el TIR.